# Python

# Документирование кода

https://numpydoc.readthedocs.io/en/latest/format.html

https://habr.com/ru/company/lamoda/blog/432656/

**Парадигма** — это стиль написания исходного кода компьютерной программы.

# Самые популярные парадигмы

- Императивное программирование
- Структурное программирование
- Декларативное программирование
- Объектно-ориентированное программирование

# Императивное программирование

**Императивное программирование** (от англ. imperative — приказ) — это парадигма программирования, которая описывает процесс вычисления в виде инструкций, **изменяющих состояние данных**. Мы описываем **КАК** выполнить задачу.

Для этой парадигмы характерно использование:

- именованных переменных;
- оператора присваивания;
- составных выражений;
- подпрограмм;
- циклов.

К подвидам императивного программирования относят **Процедурное** и **Объектно-ориентированное программирование (ООП)**.

# Декларативное программирование

**Декларативное программирование** — это парадигма программирования, в которой задается спецификация решения задачи, то есть описывается, **ЧТО** представляет собой проблема и ожидаемый результат. Декларативные программы **не используют состояния**, то есть **не содержат переменных и операторов** присваивания. Программа — спецификация описывающая решение задачи.

К подвидам декларативного программирования относят **Функциональное** и **Логическое программирование**.

# Функциональное программирование

**Функциональное программирование** — программирование значениями (не используются присваивания). Предполагает обходиться вычислением **результатов функций** от исходных данных и результатов других функций, и **не предполагает явного хранения состояния**.

Для этой парадигмы характерно:

- функции первого класса (можно передавать как аргументы и возвращать из других функций);
- функции высшего порядка (принимают на вход другие функции);
- рекурсии;
- состояние никогда не меняется;
- не используется присваивание.

Основой для функционального программирования являются **Лямбда-исчисления**, многие функциональные языки можно рассматривать как «надстройку» над ними.

**Лямбда-функции** — это функции, у которой фактически нет имени. Лямда-выражения — анонимные функции.

## Map

Принимает функцию и набор данных. Создаёт новую коллекцию, выполняет функцию на каждой позиции данных и добавляет возвращаемое значение в новую коллекцию. Возвращает новую коллекцию

Простой map, принимающий список имён и возвращающий список длин:

```
name_lengths = map(len, ['Маша', 'Петя', 'Вася'])


print name_lengths

# => [4, 4, 3]
```

Этот map возводит в квадрат каждый элемент:

```
squares = map(lambda x: x * x, [0, 1, 2, 3, 4])

print squares

# => [0, 1, 4, 9, 16]
```

Он не принимает именованную функцию, а берёт анонимную, определённую через lambda. Параметры lambda определены слева от двоеточия. Тело функции – справа. Результат возвращается неявным образом.

```
import random

names = ['Маша', 'Петя', 'Вася']

code_names = ['Шпунтик', 'Винтик', 'Фунтик']

for i in range(len(names)):
    names[i] = random.choice(code_names)

print names

# => ['Шпунтик', 'Винтик', 'Шпунтик']
```

Алгоритм может присвоить одинаковые прозвища разным секретным агентам. Будем надеяться, что это не послужит источником проблем во время секретной миссии.

```
import random


names = ['Маша', 'Петя', 'Вася']


secret_names = map(lambda x: random.choice(['Шпунтик',
'Винтик', 'Фунтик']), names)
```

## map()

```
result = map(lambda x: x ** 2,
[1,2,3,4,5,6])


print(list(result))
```

## Filter

В то время как `map()` пропускает каждый элемент итерируемого через функцию и возвращает результат всех элементов, прошедших через функцию `filter()`, прежде всего, требует, чтобы функция возвращала логические значения (true или false), а затем передает каждый элемент итерируемого через функцию, «отфильтровывая» те, которые являются ложными. Имеет следующий синтаксис:

# filter()

```
result = filter(lambda x: x % 2 == 0,
[1,2,3,4,5,6])


print(list(result))
```

Следующие пункты должны быть отмечены относительно `filter()`:

1. В отличие от `map()`, (), требуется только один итерируемый.
2. Аргумент `func` необходим для возврата логического типа. Если этого не происходит, `filter` sпросто возвращает передаваемый ему `iterable`. Кроме того, поскольку требуется только один итерируемый, подразумевается, что `func` должен принимать только один аргумент.
3. `filter` пропускает каждый элемент в итерируемом через `func` возвращает **только** только те, которые имеют значение true. Ведь это же заложено в самом названии -- «фильтр»

Ниже приведен список (`iterable`) баллов 10 студентов на экзамене по химии. Давайте отфильтруем тех, кто сдал с баллом выше 75 ... используя `filter`.

```
scores = [8, 59, 76, 60, 88, 74, 81, 65]

def is_A_student(score):

    return score > 75

over_75 = list(filter(is_A_student, scores))

 print(over_75)
```

Следующим примером будет детектор палиндрома. «Палиндром» - это слово, фраза или последовательность, которые читаются одинаково в обе стороны. Давайте отфильтруем слова, являющиеся палиндромами, из набора (`iterable`) оподозреваемых палиндромов.

```
dromes = ("demigod", "rewire", "madam", "freer", "anutforajaroftuna", "kiosk")

palindromes = list(filter(lambda word: word == word[::-1], dromes))

  print(palindromes)
```

## Reduce

`reduce` применяет функцию **двух аргументов** кумулятивно к элементам итерируемого, необязательно начиная с начального аргумента. Имеет следующий синтаксис:

```
reduce(func, iterable[, initial])
```

Где `func` это функция, к которой кумулятивно применяется каждый элемент `iterable`, а `initial` необязательное значение, которое помещается перед элементами итерируемого в вычислении и служит значением по умолчанию, когда итерируемый объект пуст. О `reduce()`должно быть отмечено следующее: 1. `func` требуется два аргумента, первый из которых является первым элементом в `iterable` (если `initial` не указан) а второй - вторым элементом в `iterable`. Если `initial` указано, то оно становится первым аргументом функции `func`, а первый элемент в `iterable` становится вторым элементом. 2. `reduce` "уменьшает" `iterable` до одного значения.

# reduce()

```
from functools import reduce
result = reduce(lambda a, x: a + x ** 2, [1,2,3], 5)
```

# Пример

```
from functools import reduce

numbers = [3, 4, 6, 9, 34, 12]

def custom_sum(first, second):

    return first + second

result = reduce(custom_sum, numbers)

  print(result)
```

Как обычно, все дело в итерациях: `reduce` берет первый и второй элементы в `numbers` и передает их соответственно в `custom_sum`. `custom_sum` вычисляет их сумму и возвращает ее в `reduce`. Затем `reduce`принимает этот результат и применяет его в качестве первого элемента к `custom_sum` и принимает следующий элемент (третий) в `numbers` в качестве второго элемента для `custom_sum`. Он делает это непрерывно (накопительно), пока `numbers` не будет исчерпан.

# Пример 2

```
from functools import reduce

numbers = [3, 4, 6, 9, 34, 12]

def custom_sum(first, second):

    return first + second

result = reduce(custom_sum, numbers, 10)

 print(result)
```

# Декораторы

Декораторы — один из самых полезных инструментов в Python

Декоратор — это функция, которая позволяет обернуть другую функцию для расширения её функциональности без непосредственного изменения её кода. Вот почему декораторы можно рассматривать как практику метапрограммирования, когда программы могут работать с другими программами как со своими данными

```
def my_decorator(func):
  def do_some_staff():
      # some action
      result = func()
      # some action
       return result
  return do_some_staff
def my_func():
  pass
my_new_func = my_decorator(my_func)
my_new_func()
```

```
def my_decorator(func):
  def do_some_staff():
      # some action
      result = func()
      # some action
       return result
  return do_some_staff
@my_decorator
def my_func():
  pass
my_func()
```

# Используем аргументы и возвращаем значения

```
def benchmark(func):
    import time
    def wrapper(*args, **kwargs):
        start = time.time()
        return_value = func(*args, **kwargs)
        end = time.time()
        print('[*] Время выполнения: {} секунд.'.format(end-start))
        return return_value
    return wrapper
```

```
@benchmark
def fetch_webpage(url):
    import requests
    webpage = requests.get(url)
    return webpage.text
webpage = fetch_webpage('https://google.com')
print(webpage)
```